Logic and Games
SS 2009

Prof. Dr. Erich Grädel
Łukasz Kaiser, Tobias Ganzow

Mathematische Grundlagen der Informatik
RWTH Aachen

Contents

1 Finite Games and First-Order Logic

An important problem in the field of logics is the question for a given logic L, a structure $\mathfrak{A}$ and a formula $\psi \in L$, whether $\mathfrak{A}$ is a model of ψ. In this chapter we will discuss an approach to the solution of this *model checking problem* via games for some logics. Our goal is to reduce the problem $\mathfrak{A} \models \psi$ to a strategy problem for a *model checking game* $\mathcal{G}(\mathfrak{A}, \psi)$ played by two players called *Verifier* (or *Player* 0) and *Falsifier* (or *Player* 1). We want to have the following relation between these two problems:

$$\mathfrak{A} \models \psi \text{ iff Verifier has a winning strategy for } \mathcal{G}(\mathfrak{A}, \psi).$$

We can then do model checking by constructing or proving the existence of winning strategies.

1.1 Model Checking Games for Modal Logic

The first logic to be considered is propositional modal logic (ML). Let us first briefly review its syntax and semantics:

Definition 1.1. For a given set of actions A and atomic properties $\{P_i : i \in I\}$, the syntax of ML is inductively defined:

- All propositional logic formulae with propositional variables P_i are in ML.
- If $\psi, \varphi \in$ ML, then also $\neg\psi$, $(\psi \wedge \varphi)$ and $(\psi \vee \varphi) \in$ ML.
- If $\psi \in$ ML and $a \in A$, then $\langle a \rangle \psi$ and $[a]\psi \in$ ML.

Remark 1.2. If there is only one action $a \in A$, we write $\Diamond\psi$ and $\Box\psi$ instead of $\langle a \rangle \psi$ and $[a]\psi$, respectively.

Definition 1.3. A *transition system* or *Kripke structure* with actions from a set A and atomic properties $\{P_i : i \in I\}$ is a structure

$$\mathcal{K} = (V, (E_a)_{a\in A}, (P_i)_{i\in I})$$

with a universe V of states, binary relations $E_a \subseteq V \times V$ describing transitions between the states, and unary relations $P_i \subseteq V$ describing the atomic properties of states.

A transition system can be seen as a labelled graph where the nodes are the states of $\mathcal{K}$, the unary relations are labels of the states, and the binary transition relations are the labelled edges.

Definition 1.4. Let $\mathcal{K} = (V, (E_a)_{a\in A}, (P_i)_{i\in I})$ be a transition system, $\psi \in \mathrm{ML}$ a formula and v a state of $\mathcal{K}$. The *model relationship* $\mathcal{K}, v \models \psi$, i.e. ψ holds at state v of $\mathcal{K}$, is inductively defined:

- $\mathcal{K}, v \models P_i$ if and only if $v \in P_i$.
- $\mathcal{K}, v \models \neg\psi$ if and only if $\mathcal{K}, v \not\models \psi$.
- $\mathcal{K}, v \models \psi \vee \varphi$ if and only if $\mathcal{K}, v \models \psi$ or $\mathcal{K}, v \models \varphi$.
- $\mathcal{K}, v \models \psi \wedge \varphi$ if and only if $\mathcal{K}, v \models \psi$ and $\mathcal{K}, v \models \varphi$.
- $\mathcal{K}, v \models \langle a\rangle\psi$ if and only if there exists w such that $(v, w) \in E_a$ and $\mathcal{K}, w \models \psi$.
- $\mathcal{K}, v \models [a]\,\psi$ if and only if $\mathcal{K}, w \models \psi$ holds for all w with $(v, w) \in E_a$.

Definition 1.5. For a transition system $\mathcal{K}$ and a formula ψ we define the *extension*

$$[\![\psi]\!]^{\mathcal{K}} := \{v : \mathcal{K}, v \models \psi\}$$

as the set of states of $\mathcal{K}$ where ψ holds.

Remark 1.6. In order to keep the following propositions short and easier to understand, we assume that all modal logic formulae are given in negation normal form, i.e. negations occur only at atoms. This does not change the expressiveness of modal logic as for every formula an equivalent one in negation normal form can be constructed. We omit a proof here, but the transformation can be easily achieved by applying

DeMorgan's laws and the duality of $\Box$ and $\Diamond$ (i.e. $\neg\langle a\rangle\psi \equiv [a]\neg\psi$ and $\neg[a]\psi \equiv \langle a\rangle\neg\psi$) to shift negations to the atomic subformulae.

We will now describe model checking games for ML. Given a transition system $\mathcal{K}$ and a formula $\psi \in \mathrm{ML}$, we define a game $\mathcal{G}$ that contains positions (φ, v) for every subformula φ of ψ and every $v \in V$. In this game, starting from position (φ, v), Verifier's goal is to show that $\mathcal{K}, v \models \varphi$, while Falsifier tries to prove $\mathcal{K}, v \not\models \varphi$.

In the game, Verifier is allowed to move at positions $(\varphi \vee \vartheta, v)$, where she can choose to move to position (φ, v) or (ϑ, v), and at positions $(\langle a\rangle\varphi, v)$, where she can move to position (φ, w) for a $w \in vE_a$. Analogously, Falsifier can move from $(\varphi \wedge \vartheta, v)$ to (φ, v) or (ϑ, v) and from $([a]\varphi, v)$ to (φ, w) for a $w \in vE_a$. Finally, there are the terminal positions (P_i, v) and $(\neg P_i, v)$, which are won by Verifier if $\mathcal{K}, v \models P_i$ and $\mathcal{K}, v \models \neg P_i$, respectively, otherwise they are winning positions for Falsifier.

The intuitive idea of this construction is to let the Verifier make the existential choices. To win from one of her positions, a disjunction or diamond subformula, she either needs to prove that one of the disjuncts is true, or that there exists a successor at which the subformula holds. Falsifier, on the other hand, in order to win from his positions, can choose a conjunct that is false or, if at a box formula, choose a successor at which the subformula does not hold.

The idea behind this construction is that at disjunctions and diamonds, Verifier can choose a subformula that is satisfied by the structure or a successor position at which the subformula is satisfied, while at conjunctions and boxes, Falsifier can choose a subformula or position that is not. So it is easy to see that the following lemma holds.

Lemma 1.7. Let $\mathcal{K}$ be a Kripke structure, $v \in V$ and φ a formula in ML. Then we have

$$\mathcal{K}, v \models \varphi \quad \Leftrightarrow \quad \text{Verifier has a winning strategy from } (\varphi, v).$$

To assess the efficiency of games as a solution for model checking problems, we have to consider the complexity of the resulting model checking games based on the following criteria:

- Are all plays necessarily finite?
- If not, what are the winning conditions for infinite plays?
- Do the players always have perfect information?
- What is the structural complexity of the game graphs?
- How does the size of the graph depend on different parameters of the input structure and the formula?

For first-order logic (FO) and modal logic (ML) we have only finite plays with positional winning conditions, and, as we will see, the winning regions are computable in linear time with respect to the size of the game graph (for finite structures of course).

Model checking games for fixed-point logics however admit infinite plays, and we use so called *parity conditions* to determine the winner of such plays. It is still an open question whether winning regions and winning strategies in parity games are computable in polynomial time.

1.2 Finite Games

In the following section we want to deal with two-player games with perfect information and positional winning conditions, given by a *game graph* (or *arena*)

$$\mathcal{G} = (V, E)$$

where the set V of positions is partitioned into sets of positions V_0 and V_1 belonging to Player 0 and Player 1, respectively. Player 0, also called *Ego*, moves from positions $v \in V_0$, while Player 1, called *Alter*, moves from positions $v \in V_1$. All moves are along edges, and we use the term *play* to describe a (finite or infinite) sequence $v_0 v_1 v_2 \dots$ with $(v_i, v_{i+1}) \in E$ for all i. We use a simple positional winning condition: Move or lose! Player σ wins at position v if $v \in V_{1-\sigma}$ and $vE = \emptyset$, i.e., if the position belongs to his opponent and there are no moves possible from that position. Note that this winning condition only applies to finite plays, infinite plays are considered to be a draw.

We define a *strategy* (for Player σ) as a mapping

$$f : \{v \in V_\sigma : vE \neq \emptyset\} \to V$$

with $(v, f(v)) \in E$ for all $v \in V$. We call f *winning* from position v if Player σ wins all plays that start at v and are consistent with f.

We now can define *winning regions* W_0 and W_1:

$$W_\sigma = \{v \in V : \text{Player } \sigma \text{ has a winning strategy from position } v\}.$$

This proposes several algorithmic problems for a given game $\mathcal{G}$: The computation of winning regions W_0 and W_1, the computation of winning strategies, and the associated decision problem

$$\textsc{Game} := \{(\mathcal{G}, v) : \text{Player 0 has a winning strategy for } \mathcal{G} \text{ from } v\}.$$

Theorem 1.8. Game is P-complete and decidable in time $O(|V| + |E|)$.

Note that this remains true for *strictly alternating games*.

A simple polynomial-time approach to solve Game is to compute the winning regions inductively: $W_\sigma = \bigcup_{n \in \mathbb{N}} W_\sigma^n$, where

$$W_\sigma^0 = \{v \in V_{1-\sigma} : vE = \emptyset\}$$

is the set of terminal positions which are winning for Player σ, and

$$W_\sigma^{n+1} = \{v \in V_\sigma : vE \cap W_\sigma^n \neq \emptyset\} \cup \{v \in V_{1-\sigma} : vE \subseteq W_\sigma^n\}$$

is the set of positions from which Player σ can win in at most $n+1$ moves.

After $n \leq |V|$ steps, we have that $W_\sigma^{n+1} = W_\sigma^n$, and we can stop the computation here.

To solve Game in linear time, we have to use the slightly more involved Algorithm 1.1. Procedure Propagate will be called once for every edge in the game graph, so the running time of this algorithm is linear with respect to the number of edges in $\mathcal{G}$.

Furthermore, we can show that the decision problem Game is equivalent to the satisfiability problem for propositional Horn formulae.

Algorithm 1.1. A linear time algorithm for Game

```
Input: A game $\mathcal{G} = (V, V_0, V_1, E)$
output: Winning regions $W_0$ and $W_1$

for all $v \in V$ do                 (∗ 1: Initialisation ∗)
    win[$v$] := $\bot$
    $P[v] := \emptyset$
    $n[v] := 0$
end do

for all $(u, v) \in E$ do            (∗ 2: Calculate $P$ and $n$ ∗)
    $P[v] := P[v] \cup \{u\}$
    $n[u] := n[u] + 1$
end do

for all $v \in V_0$                  (∗ 3: Calculate win ∗)
    if $n[v] = 0$ then Propagate($v$, 1)
for all $v \in V \setminus V_0$
    if $n[v] = 0$ then Propagate($v$, 0)
return win

procedure Propagate($v$, $\sigma$)
    if win[$v$] $\neq \bot$ then return
    win[$v$] := $\sigma$             (∗ 4: Mark $v$ as winning for player $\sigma$ ∗)
    for all $u \in P[v]$ do          (∗ 5: Propagate change to predecessors ∗)
        $n[u] := n[u] - 1$
        if $u \in V_\sigma$ or $n[u] = 0$ then Propagate($u$, $\sigma$)
    end do
end
```

We recall that propositional Horn formulae are finite conjunctions $\bigwedge_{i \in I} C_i$ of clauses C_i of the form

$$\begin{aligned} X_1 \land \ldots \land X_n &\rightarrow X \quad \text{or} \\ \underbrace{X_1 \land \ldots \land X_n}_{\text{body}(C_i)} &\rightarrow \underbrace{0}_{\text{head}(C_i)} . \end{aligned}$$

A clause of the form X or $1 \rightarrow X$ has an empty body.

We will show that Sat-Horn and Game are mutually reducible via logspace and linear-time reductions.

(1) Game $\leq_{\text{log-lin}}$ Sat-Horn
For a game $\mathcal{G} = (V, V_0, V_1, E)$, we construct a Horn formula $\psi_{\mathcal{G}}$ with clauses

$$\begin{aligned} v \rightarrow u \quad & \text{for all } u \in V_0 \text{ and } (u, v) \in E\text{, and} \\ v_1 \land \ldots \land v_m \rightarrow u \quad & \text{for all } u \in V_1 \text{ and } uE = \{v_1, \ldots, v_m\}. \end{aligned}$$

The minimal model of $\psi_{\mathcal{G}}$ is precisely the winning region of Player 0, so

$$(\mathcal{G}, v) \in \text{Game} \iff \psi_{\mathcal{G}} \land (v \rightarrow 0) \text{ is unsatisfiable.}$$

(2) Sat-Horn $\leq_{\text{log-lin}}$ Game
For a Horn formula $\psi(X_1, \ldots, X_n) = \bigwedge_{i \in I} C_i$, we define a game $\mathcal{G}_\psi = (V, V_0, V_1, E)$ as follows:

$$V = \underbrace{\{0\} \cup \{X_1, \ldots, X_n\}}_{V_0} \cup \underbrace{\{C_i : i \in I\}}_{V_1} \text{ and}$$

$$E = \{X_j \rightarrow C_i : X_j = \text{head}(C_i)\} \cup \{C_i \rightarrow X_j : X_j \in \text{body}(C_i)\},$$

i.e., Player 0 moves from a variable to some clause containing the variable as its head, and Player 1 moves from a clause to some variable in its body. Player 0 wins a play if, and only if, the play reaches a clause C with body$(C) = \emptyset$. Furthermore, Player 0 has a winning strategy from position X if, and only if, $\psi \models X$, so we have

$$\text{Player 0 wins from position 0} \iff \psi \text{ is unsatisfiable.}$$

These reductions show that SAT-HORN is also P-complete and, in particular, also decidable in linear time.

1.3 Alternating Algorithms

Alternating algorithms are algorithms whose set of configurations is divided into *accepting*, *rejecting*, *existential* and *universal* configurations. The acceptance condition of an alternating algorithm A is defined by a game played by two players $\exists$ and $\forall$ on the computation tree $\mathcal{T}_{A,x}$ of A on input x. The positions in this game are the configurations of A, and we allow moves $C \to C'$ from a configuration C to any of its successor configurations C'. Player $\exists$ moves at existential configurations and wins at accepting configurations, while Player $\forall$ moves at universal configurations and wins at rejecting configurations. By definition, A accepts some input x if and only if Player $\exists$ has a winning strategy for the game played on $\mathcal{T}_{A,x}$.

We will introduce the concept of alternating algorithms formally, using the model of a Turing machine, and we prove certain relationships between the resulting alternating complexity classes and usual deterministic complexity classes.

1.3.1 Turing Machines

The notion of an alternating Turing machine extends the usual model of a (deterministic) Turing machine which we introduce first. We consider Turing machines with a separate input tape and multiple linear work tapes which are divided into basic units, called cells or fields. Informally, the Turing machine has a reading head on the input tape and a combined reading and writing head on each of its work tapes. Each of the heads is at one particular cell of the corresponding tape during each point of a computation. Moreover, the Turing machine is in a certain state. Depending on this state and the symbols the machine

is currently reading on the input and work tapes, it manipulates the current fields of the work tapes, moves its heads and changes to a new state.

Formally, a *(deterministic) Turing machine with separate input tape and k linear work tapes* is given by a tuple $M = (Q, \Gamma, \Sigma, q_0, F_{\text{acc}}, F_{\text{rej}}, \delta)$, where Q is a finite set of *states*, Σ is the *work alphabet* containing a designated symbol $\square$ (*blank*), Γ is the *input alphabet*, $q_0 \in Q$ is the *initial state*, $F := F_{\text{acc}} \cup F_{\text{rej}} \subseteq Q$ is the set of *final states* (with F_{acc} the *accepting states*, F_{rej} the *rejecting states* and $F_{\text{acc}} \cap F_{\text{rej}} = \emptyset$), and $\delta : (Q \setminus F) \times \Gamma \times \Sigma^k \to Q \times \{-1,0,1\} \times \Sigma^k \times \{-1,0,1\}^k$ is the *transition function*.

A *configuration* of M is a complete description of all relevant facts about the machine at some point during a computation, so it is a tuple $C = (q, w_1, \dots, w_k, x, p_0, p_1, \dots, p_k) \in Q \times (\Sigma^*)^k \times \Gamma^* \times \mathbb{N}^{k+1}$ where q is the recent state, w_i is the contents of work tape number i, x is the contents of the input tape, p_0 is the position on the input tape and p_i is the position on work tape number i. The contents of each of the tapes is represented as a finite word over the corresponding alphabet[, i.e., a finite sequence of symbols from the alphabet]. The contents of each of the fields with numbers $j > |w_i|$ on work tape number i is the blank symbol (we think of the tape as being infinite). A configuration where x is omitted is called a *partial configuration*. The configuration C is called *final* if $q \in F$. It is called *accepting* if $q \in F_{\text{acc}}$ and *rejecting* if $q \in F_{\text{rej}}$.

The *successor configuration* of C is determined by the recent state and the $k+1$ symbols on the recent cells of the tapes, using the transition function: If $\delta(q, x_{p_0}, (w_1)_{p_1}, \dots, (w_k)_{p_k}) = (q', m_0, a_1, \dots, a_k, m_1, \dots, m_k, b)$, then the successor configuration of C is $\Delta(C) = (q', \overline{w}', \overline{p}', x)$, where for any i, w_i' is obtained from w_i by replacing symbol number p_i by a_i and $p_i' = p_i + m_i$. We write $C \vdash_M C'$ if, and only if, $C' = \Delta(C)$.

The *initial configuration* $C_0(x) = C_0(M, x)$ of M on input $x \in \Gamma^*$ is given by the initial state q_0, the blank-padded memory, i.e., $w_i = \varepsilon$ and $p_i = 0$ for any $i \geq 1$, $p_0 = 0$, and the contents x on the input tape.

A *computation* of M on input x is a sequence $C_0, C_1, \dots$ of configurations of M, such that $C_0 = C_0(x)$ and $C_i \vdash_M C_{i+1}$ for all $i \geq 0$. The computation is called *complete* if it is infinite or ends in some final configuration. A complete finite computation is called *accepting* if the last configuration is accepting, and the computation is called *rejecting* if the last configuration is rejecting. *M accepts* input x if the (unique) complete computation of M on x is finite and accepting. *M rejects* input x if the (unique) complete computation of M on x is finite and rejecting. The machine M *decides* a language $L \subseteq \Gamma^*$ if M accepts all $x \in L$ and rejects all $x \in \Gamma^* \setminus L$.

1.3.2 Alternating Turing Machines

Now we shall extend deterministic Turing machines to nondeterministic Turing machines from which the concept of alternating Turing machines is obtained in a very natural way, given our game theoretical framework.

A *nondeterministic Turing machine* is nondeterministic in the sense that a given configuration C may have several possible successor configurations instead of at most one. Intuitively, this can be described as the ability to *guess*. This is formalised by replacing the transition function $\delta : (Q \setminus F) \times \Gamma \times \Sigma^k \to Q \times \{-1, 0, 1\} \times \Sigma^k \times \{-1, 0, 1\}^k$ by a transition relation $\Delta \subseteq ((Q \setminus F) \times \Gamma \times \Sigma^k) \times (Q \times \{-1, 0, 1\} \times \Sigma^k \times \{-1, 0, 1\}^k)$. The notion of successor configurations is defined as in the deterministic case, except that the successor configuration of a configuration C may not be uniquely determined. Computations and all related notions carry over from deterministic machines in the obvious way. However, on a fixed input x, a nondeterministic machine now has several possible computations, which form a (possibly infinite) finitely branching computation tree $\mathcal{T}_{M,x}$. A nondeterministic Turing machine M *accepts* an input x if there *exists* a computation of M on x which is accepting, i.e., if there exists a path from the root $C_0(x)$ of $\mathcal{T}_{M,x}$ to some accepting configuration. The language of M is $L(M) = \{x \in \Gamma^* \mid M \text{ accepts } x\}$. Notice that for a nondeterministic machine M to decide a language $L \subseteq \Gamma^*$ it is not necessary, that all computations of M are finite. (In a sense, we count infinite computations as rejecting.)

From a game-theoretical perspective, the computation of a nondeterministic machine can be viewed as a solitaire game on the computation tree in which the only player (the machine) chooses a path

through the tree starting from the initial configuration. The player wins the game (and hence, the machine accepts its input) if the chosen path finally reaches an accepting configuration.

An obvious generalisation of this game is to turn it into a two-player game by assigning the nodes to the two players who are called $\exists$ and $\forall$, following the intuition that Player $\exists$ tries to show the existence of a *good* path, whereas Player $\forall$ tries to show that all selected paths are *bad*. As before, Player $\exists$ wins a play of the resulting game if, and only if, the play is finite and ends in an accepting leaf of the game tree. Hence, we call a computation tree accepting if, and only if, Player $\exists$ has a winning strategy for this game.

It is important to note that the partition of the nodes in the tree should not depend on the input x but is supposed to be inherent to the machine. Actually, it is even independent of the contents of the work tapes, and thus, whether a configuration belongs to Player $\exists$ or to Player $\forall$ merely depends on the current state.

Formally, an *alternating Turing machine* is a nondeterministic Turing machine $M = (Q, \Gamma, \Sigma, q_0, F_{\text{acc}}, F_{\text{rej}}, \Delta)$ whose set of states $Q = Q_\exists \cup Q_\forall \cup F_{\text{acc}} \cup F_{\text{rej}}$ is partitioned into *existential, universal, accepting*, and *rejecting* states. The semantics of these machines is given by means of the game described above.

Now, if we let accepting configurations belong to player $\forall$ and rejecting configurations belong to player $\exists$, then we have the usual winning condition that a player loses if it is his turn but he cannot move. We can solve such games by determining the winner at leaf nodes and propagating the winner successively to parent nodes. If at some node, the winner at all of its child nodes is determined, the winner at this node can be determined as well. This method is sometimes referred to as backwards induction and it basically coincides with our method for solving Game on trees (with possibly infinite plays). This gives the following equivalent semantics of alternating Turing machines:

The subtree $\mathcal{T}_C$ of the computation tree of M on x with root C is called *accepting*, if

- C is accepting

- C is existential and there is a successor configuration C' of C such that $\mathcal{T}_{C'}$ is accepting or
- C is universal and $\mathcal{T}_{C'}$ is accepting for all successor configurations C' of C.

M accepts an input x, if $\mathcal{T}_{C_0(x)} = \mathcal{T}_{M,x}$ is accepting.

For functions $T, S : \mathbb{N} \to \mathbb{N}$, an alternating Turing machine M is called *T-time bounded* if, and only if, for any input x, each computation of M on x has length less or equal $T(|x|)$. The machine is called *S-space bounded* if, and only if, for any input x, during any computation of M on x, at most $S(|x|)$ cells of the work tapes are used. Notice that time boundedness implies finiteness of all computations which is not the case for space boundedness. The same definitions apply for deterministic and nondeterministic Turing machines as well since these are just special cases of alternating Turing machines. These notions of resource bounds induce the complexity classes Atime containing precisely those languages L such that there is an alternating T-time bounded Turing machine deciding L and Aspace containing precisely those languages L such that there is an alternating S-space bounded Turing machine deciding L. Similarly, these classes can be defined for nondeterministic and deterministic Turing machines.

We are especially interested in the following alternating complexity classes:

- ALogspace $= \bigcup_{d\in\mathbb{N}}$ Aspace$(d \cdot \log n)$,
- APtime $= \bigcup_{d\in\mathbb{N}}$ Atime(n^d),
- APspace $= \bigcup_{d\in\mathbb{N}}$ Aspace(n^d).

Observe that Game $\in$ Alogspace. An alternating algorithm which decides Game with logarithmic space just plays the game. The algorithm only has to store the *current* position in memory, and this can be done with logarithmic space. We shall now consider a slightly more involved example.

Example 1.9. QBF $\in$ Atime(O(n)). W.l.o.g we assume that negation appears only at literals. We describe an alternating procedure $Eval(\varphi, \mathcal{I})$ which computes, given a quantified Boolean formula ψ and a valuation $\mathcal{I} : \text{free}(\psi) \to \{0,1\}$ of the free variables of ψ, the value $[\![\psi]\!]^{\mathcal{I}}$.

Algorithm 1.2. Alternating algorithm deciding QBF.

```
Input: $(\psi, \mathcal{I})$   where $\psi \in$ QAL and $\mathcal{I}$: free$(\psi) \to \{0,1\}$
if $\psi = Y$          then
                          if $\mathcal{I}(Y) = 1$ then accept
                                        else reject
if $\psi = \varphi_1 \vee \varphi_2$  then „∃“ guesses $i \in \{1,2\}$ , $Eval(\varphi_i, \mathcal{I})$
if $\psi = \varphi_1 \wedge \varphi_2$  then „∀“ chooses $i \in \{1,2\}$ , $Eval(\varphi_i, \mathcal{I})$
if $\psi = \exists X\varphi$      then „∃“ guesses $j \in \{0,1\}$ , $Eval(\varphi, \mathcal{I}[X = j])$
if $\psi = \forall X\varphi$      then „∀“ chooses $j \in \{0,1\}$ , $Eval(\varphi, \mathcal{I}[X = j])$
```

The main results we want to establish in this section concern the relationship between alternating complexity classes and deterministic complexity classes. We will see that alternating time corresponds to deterministic space, while by translating deterministic time into alternating space, we can reduce the complexity by one exponential. Here, we consider the special case of alternating polynomial time and polynomial space. We should mention, however, that these results can be generalised to arbitrary large complexity bounds which are well behaved in a certain sense.

Lemma 1.10. NPspace $\subseteq$ APtime.

Proof. Let $L \in$ NPspace and let M be a nondeterministic n^l-space bounded Turing machine which recognises L for some $l \in \mathbb{N}$. The machine M accepts some input x if, and only if, some accepting configuration is reachable from the initial configuration $C_0(x)$ in the configuration tree of M on x in at most $k := 2^{cn^l}$ steps for some $c \in \mathbb{N}$. This is due to the fact that there are most k different configurations of M on input x which use at most n^l cells of the memory which can be seen using a simple combinatorial argument. So if there is some accepting configuration reachable from the initial configuration $C_0(x)$, then there is some accepting configuration reachable from $C_0(x)$ in at most k steps. This is equivalent to the existence of some intermediate configuration C' that is reachable from $C_0(x)$ in at most $k/2$ steps and from which some accepting configuration is reachable in at most $k/2$ steps.

So the alternating algorithm deciding L proceeds as follows. The existential player guesses such a configuration C' and the universal player chooses whether to check that C' is reachable from $C_0(x)$ in at most $k/2$ steps or whether to check that some accepting configuration is reachable from C' in at most $k/2$ steps. Then the algorithm (or equivalently, the game) proceeds with the subproblem chosen by the universal player, and continues in this binary search like fashion. Obviously, the number of steps which have to be performed by this procedure to decide whether x is accepted by M is logarithmic in k. Since k is exponential in n^l, the time bound of M is dn^l for some $d \in \mathbb{N}$, so M decides L in polynomial time. Q.E.D.

Lemma 1.11. APTIME $\subseteq$ PSPACE.

Proof. Let $L \in$ APTIME and let A be an alternating n^l-time bounded Turing machine that decides L for some $l \in \mathbb{N}$. Then there is some $r \in \mathbb{N}$ such that any configuration of A on any input x has at most r successor configurations and w.l.o.g. we can assume that any non-final configuration has precisely r successor configurations. We can think of the successor configurations of some non-final configuration C as being enumerated as $C_1, \dots, C_r$. Clearly, for given C and i we can compute C_i. The idea for a deterministic Turing machine M to check whether some input x is in L is to perform a depth-first search on the computation tree $\mathcal{T}_{A,x}$ of A on x. The crucial point is, that we cannot construct and keep the whole configuration tree $\mathcal{T}_{A,x}$ in memory since its size is exponential in $|x|$ which exceeds our desired space bound. However, since the length of each computation is polynomially bounded, it is possible to keep a single computation path in memory and to construct the successor configurations of the configuration under consideration on the fly.

Roughly, the procedure M can be described as follows. We start with the initial configuration $C_0(x)$. Given any configuration C under consideration, we propagate 0 to the predecessor configuration if C is rejecting and we propagate 1 to the predecessor configuration if C is accepting. If C is neither accepting nor rejecting, then we construct, for $i = 1, \dots, r$ the successor configuration C_i of C and proceed with checking C_i. If C is existential, then as soon as we receive 1 for some i, we propagate 1 to the predecessor. If we encounter 0 for all i, then we propagate 0. Analogously, if C is universal, then as soon as we receive a 0 for some i, we propagate 0. If we receive only 1 for all i, then we propagate 1. Then x is in L if, and only if, we finally receive 1 at $C_0(x)$. Now, at any point during such a computation we have to store at most one complete computation of A on x. Since A is n^l-time bounded, each such computation has length at most n^l and each configuration has size at most $c \cdot n^l$ for some $c \in \mathbb{N}$. So M needs at most $c \cdot n^{2l}$ memory cells which is polynomial in n. Q.E.D.

So we obtain the following result.

Theorem 1.12. (Parallel time complexity = sequential space complexity)

(1) APTIME $=$ PSPACE.
(2) AEXPTIME $=$ EXPSPACE.

Proposition (2) of this theorem is proved exactly the same way as we have done it for proposition (1). Now we prove that by translating sequential *time* into alternating *space*, we can reduce the complexity by one exponential.

Lemma 1.13. EXPTIME $\subseteq$ APSPACE

Proof. Let $L \in$ EXPTIME. Using a standard argument from complexity theory, there is a deterministic Turing machine $M = (Q, \Sigma, q_0, \delta)$ with time bound $m := 2^{c \cdot n^k}$ for some $c, k \in \mathbb{N}$ with only a single tape (serving as both input and work tape) which decides L. (The time bound of the machine with only a single tape is quadratic in that of the original machine with k work tapes and a separate input tape, which, however, does not matter in the case of an exponential time bound.) Now if $\Gamma = \Sigma \uplus (Q \times \Sigma) \uplus \{\#\}$, then we can describe each configuration C of M by a word

$$\underline{C} = \# w_0 \dots w_{i-1} (q w_i) w_{i+1} \dots w_t \# \in \Gamma^*.$$

Since M has time bound m and only one single tape, it has space bound m. So, w.l.o.g., we can assume that $|\underline{C}| = m + 2$ for all configurations C of M on inputs of length n. (We just use a representation of the tape which has a priori the maximum length that will occur during a computation on an input of length n.) Now the crucial point in the argumentation is the following. If $C \vdash C'$ and $1 \leq i \leq m$, symbol number i of the word $\underline{C'}$ only depends on the symbols number $i - 1$, i and $i + 1$ of $\underline{C}$. This allows us, to decide whether $x \in L(M)$ with the following alternating procedure which uses only polynomial space.

Player $\exists$ guesses some number $s \leq m$ of steps of which he claims that it is precisely the length of the computation of M on input x. Furthermore, $\exists$ guesses some state $q \in F_{\text{acc}}$, a Symbol $a \in \Sigma$ and a number $i \in \{0, \ldots, s\}$, and he claims that the i-th symbol of the configuration $\underline{C}$ of M after the computation on x is (qa). (So players start inspecting the computation of M on x from the final configuration.) If M accepts input x, then obviously player $\exists$ has a possibility to choose all these objects such that his claims can be validated. Player $\forall$ wants to disprove the claims of $\exists$. Now, player $\exists$ guesses symbols $a_{-1}, a_0, a_1 \in \Gamma$ of which he claims that these are the symbols number $i - 1$, i and $i + 1$ of the predecessor configuration of the final configuration $\underline{C}$. Now, $\forall$ can choose any of these symbols and demand, that $\exists$ validates his claim for this particular symbol. This symbol is now the symbol under consideration, while i is updated according to the movement of the (unique) head of M. Now, these actions of the players take place for each of the s computation steps of M on x. After s such steps, we check whether the recent symbol and the recent position are consistent with the initial configuration $C_0(x)$. The only information that has to be stored in the memory is the position i on the tape, the number s which $\exists$ has initially guessed and the current number of steps. Therefore, the algorithm uses space at most $\mathrm{O}(\log(m)) = \mathrm{O}(n^k)$ which is polynomial in n. Moreover, if M accepts input x then obviously, player $\exists$ has a winning strategy for the computation game. If, conversely, M rejects input x, then the combination of all claims of player $\exists$ cannot be consistent and player $\forall$ has a strategy to spoil any (cheating) strategy of player $\exists$ by choosing the appropriate symbol at the appropriate

computation step. Q.E.D.

Finally, we make the simple observation that it is not possible to gain more than one exponential when translating from sequential time to alternating space. (Notice that EXPTIME is a proper subclass of 2EXPTIME.)

Lemma 1.14. APSPACE $\subseteq$ EXPTIME

Proof. Let $L \in$ APSPACE, and let A be an alternating n^k-space bounded Turing machine which decides L for some $k \in \mathbb{N}$. Moreover, for an input x of A, let $\mathrm{Conf}(A, x)$ be the set of all configurations of A on input x. Due to the polynomial space bound of A, this set is finite and its size is at most exponential in $|x|$. So we can construct the graph $G = (\mathrm{Conf}(A, x), \vdash)$ in time exponential in $|x|$. Moreover, a configuration C is reachable from $C_0(x)$ in $\mathcal{T}_{A,x}$ if and only if C is reachable from $C_0(x)$ in G. So to check whether A accepts input x we simply decide whether player $\exists$ has a winning strategy for the game played on G from $C_0(x)$. This can be done in time linear in the size of G, so altogether we can decide whether $x \in L(A)$ in time exponential in $|x|$. Q.E.D.

Theorem 1.15. (Translating sequential time into alternating space)

(1) ALOGSPACE = P.
(2) APSPACE = EXPTIME.

Proposition (1) of this theorem is proved using exactly the same arguments as we have used for proving proposition (2). An overview over the relationship between deterministic and alternating complexity classes is given in Figure 1.1.

$$\begin{array}{ccccccccc}
\textsc{Logspace} & \subseteq & \textsc{Ptime} & \subseteq & \textsc{Pspace} & \subseteq & \textsc{Exptime} & \subseteq & \textsc{Expspace} \\
 & & \| & & \| & & \| & & \| \\
 & & \textsc{ALogspace} & \subseteq & \textsc{APtime} & \subseteq & \textsc{APspace} & \subseteq & \textsc{AExptime}
\end{array}$$

Figure 1.1. Relation between deterministic and alternating complexity classes

1.4 Model Checking Games for First-Order Logic

Let us first recall the syntax of FO formulae on relational structures. We have that $R_i(\bar{x})$, $\neg R_i(\bar{x})$, $x = y$ and $x \neq y$ are well-formed valid FO formulae, and inductively for FO formulae φ and ψ, we have that $\varphi \vee \psi$, $\varphi \wedge \psi$, $\exists x \varphi$ and $\forall x \varphi$ are well-formed FO formulae. This way, we allow only formulae in *negation normal form* where negations occur only at atomic subformulae and all junctions except $\vee$ and $\wedge$ are eliminated. These constraints do not limit the expressiveness of the logic, but the resulting games are easier to handle.

For a structure $\mathfrak{A} = (A, R_1, \dots, R_m)$ with $R_i \subseteq A^{r_i}$, we define the evaluation game $\mathcal{G}(\mathfrak{A}, \psi)$ as follows:

We have positions $\varphi(\bar{a})$ for every subformula $\varphi(\bar{x})$ of ψ and every $\bar{a} \in A^k$.

At a position $\varphi \vee \vartheta$, Verifier can choose to move either to φ or to ϑ, while at positions $\exists x \varphi(x, \bar{b})$, he can choose an instantiation $a \in A$ of x and move to $\varphi(a, \bar{b})$. Analogously, Falsifier can move from positions $\varphi \wedge \vartheta$ to either φ or ϑ and from positions $\forall x \varphi(x, \bar{b})$ to $\varphi(a, \bar{b})$ for an $a \in A$.

The winning condition is evaluated at positions with atomic or negated atomic formulae φ, and we define that Verifier wins at $\varphi(\bar{a})$ if, and only if, $\mathfrak{A} \models \varphi(\bar{a})$, and Falsifier wins if, and only if, $\mathfrak{A} \not\models \varphi(\bar{a})$.

In order to determine the complexity of FO model checking, we have to consider the process of determining whether $\mathfrak{A} \models \psi$. To decide this question, we have to construct the game $\mathcal{G}(\mathfrak{A}, \psi)$ and check whether Verifier has a winning strategy from position ψ. The size of the game graph is bound by $|\mathcal{G}(\mathfrak{A}, \psi)| \leq |\psi| \cdot |A|^{\text{width}(\psi)}$, where $\text{width}(\psi)$ is the maximal number of free variables in the subformulae of ψ. So the game graph can be exponential, and therefore we can get only exponential time complexity for Game. In particular, we have the following complexities for the general case:

- alternating time: $O(|\psi| + \text{qd}(\psi) \log |A|)$
  where $\text{qd}(\psi)$ is the quantifier-depth of ψ,
- alternating space: $O(\text{width}(\psi) \cdot \log |A| + \log |\psi|)$,
- deterministic time: $O(|\psi| \cdot |A|^{\text{width}(\psi)})$ and

- deterministic space: $O(|\psi| + \text{qd}(\psi) \log |A|)$.

Efficient implementations of model checking algorithms will construct the game graph on the fly while solving the game.

There are several possibilities of how to reason about the complexity of FO model checking. We can consider the *structural complexity*, i.e., we fix a formula and measure the complexity of the model checking algorithm in terms of the size of the structure only. On the other hand, the *expression complexity* measures the complexity in terms of the size of a given formula while the structure is considered to be fixed. Finally, the *combined complexity* is determined by considering both, the formula and the structure, as input parameters.

We obtain that the structural complexity of FO model checking is ALogtime, and both the expression complexity and the combined complexity is PSpace.

1.4.1 *Fragments of* FO *with Efficient Model Checking*

We have just seen that in the general case the complexity of FO model checking is exponential with respect to the width of the formula. In this section, we will see that some restrictions made to the underlying logic will also reduce the complexity of the associated model checking problem.

We will start by considering the *k-variable fragment of* FO :

$$\text{FO}^k := \{\psi \in \text{FO} : \text{width}(\psi) \leq k\}.$$

In this fragment, we have an upper bound for the width of the formulae, and we get polynomial time complexity:

ModCheck(FO^k) is P-complete and solvable in time $O(|\psi| \cdot |A|^k)$.

There are other fragments of FO that have model checking complexity $O(|\psi| \cdot \|\mathfrak{A}\|)$:

- ML: propositional modal logic,
- FO^2: formulae of width two,
- GF: the guarded fragment of first-order logic.

We will have a closer look at the last one, GF.

GF is a fragment of first-order logic which allows only guarded quantification

$$\exists \bar{y}(\alpha(\bar{x},\bar{y}) \wedge \varphi(\bar{x},\bar{y})) \text{ and } \forall \bar{y}(\alpha(\bar{x},\bar{y}) \rightarrow \varphi(\bar{x},\bar{y}))$$

where the *guards* α are atomic formulae containing all free variables of φ.

GF is a generalisation of modal logics, and we have that ML $\subseteq$ GF $\subseteq$ FO. In particular, the modal logic quantifiers $\Diamond$ and $\Box$ can be expressed as

$$\langle a \rangle \varphi \equiv \exists y(E_a xy \wedge \varphi(y)) \text{ and } [a]\varphi \equiv \forall y(E_a xy \rightarrow \varphi(y)).$$

Since guarded logics have small model checking games of size $\|\mathcal{G}(\mathfrak{A},\psi)\| = O(|\psi| \cdot \|\mathfrak{A}\|)$, there exist efficient game-based model checking algorithms for them.